**1. План урока**

## На уроке

1. Рассмотрите, что такое **логирование**. Логирование — это процесс записи информации о работе приложения, то есть сообщения об ошибках, предупреждения и другая отладочная информация.
2. **Узнаете, как использовать библиотеку**

logging

 в Python для вывода логов. Логи помогают разработчикам понимать, что происходит в их приложении и какие проблемы могут возникнуть.

1. Изучите различные **уровни логирования**, от

DEBUG

 до

CRITICAL

. Узнаете, как использовать уровни логирования для отладки приложений, чтобы видеть только необходимую информацию в зависимости от ситуации.

1. Познакомитесь с **основами логирования** и **компонентами библиотеки**

logging

 — это логеры, хендлеры и форматеры. Логеры управляют потоком логов, хендлеры направляют логи в нужные места, например файлы или консоль, а форматеры задают формат сообщений.

1. Рассмотрите, как создавать **логеры** и настраивать их для вывода логов в файлы и консоль. Это позволяет организовать удобное хранение и просмотр логов.
2. Научитесь создавать различные **хендлеры** для логов и применять их к логерам. Хендлеры отвечают за направление логов в нужные места, такие как файлы, консоль или удаленные серверы. Это позволяет организовать централизованное или распределенное логирование в зависимости от потребностей вашего приложения.
3. Научитесь создавать логи различного вида с помощью настройки параметров **форматеров**. Это поможет вам сделать логи более информативными и понятными для различных групп пользователей, будь то разработчики, администраторы или аналитики.

## Про домашку

Продолжите работать с проектом. Настроите логирование работы отдельных модулей вашей программы, чтобы сделать отладку работы программы более простой, а также отслеживать состояние вашей программы в процессе работы.

# 2. Что такое логирование

При разработке приложений часто требуется логирование.

**Логирование в Python** — это процесс записи информации о том, что происходит в программе во время ее работы.

Логи представляют собой сообщения или записи о событиях, которые могут быть полезны:

* при отладке программы,
* мониторинге работы программы,
* анализе работы программы.

Вы могли использовать для вывода в консоль информации о работе вашей программы функцию

print()

. Но функция

print()

 имеет ограниченные возможности. Библиотека

logging

 предоставляет более мощные и гибкие механизмы для записи логов, которые помогают разработчикам лучше контролировать и анализировать выполнение приложений.

Посмотрим, как выводится информация с помощью функции

print()

 и с использованием библиотеки

logging

.

Код с

print()

:

name = "Alice"

age = 30

# Выводим сообщение в консоль

print("Имя: " + name + ", возраст: " + str(age))

Код с

logging

:

import logging

name = "Alice"

age = 30

# Выводим сообщение в лог

logging.info (f"Имя: {name}, возраст: {age}")

logging

 — библиотека Python, которая позволяет записывать сообщения в лог-файлы, чтобы отслеживать работу приложения.

### Преимущества библиотеки logging

Логирование с помощью библиотеки

logging

 имеет множество преимуществ:

1. **Гибкость записи данных.** В отличие от функции

print()

, которая выводит данные только в консоль, библиотека

logging

 позволяет записывать данные в различные источники, включая файлы, базы данных и системы мониторинга.

1. **Контроль уровня сообщений.** Библиотека

logging

 позволяет настраивать уровни логирования, выбирая только важные сообщения для записи. Это позволяет разрабатывать приложения в среде реальных данных без засорения журналов ненужными сообщениями.

1. **Форматирование логов.** Библиотека предоставляет возможность добавлять к сообщениям дополнительную информацию, такую как дата и время, имя функции, уровень сообщения и т. д., что делает логи более читабельными и полезными.
2. **Интеграция с другими инструментами.** Библиотека

logging

 легко интегрируется с системами мониторинга и анализа логов, что позволяет автоматизировать оповещения о проблемах и анализировать производительность приложения.

# 3. Основы логирования

Познакомимся ближе с библиотекой

logging

. Для начала рассмотрим уровни логирования, которые помогают разделять логи по важности.

## Уровни логирования

На основе серьезности сообщений выделяется пять уровней логирования :

1. DEBUG

 — сообщения для отладки приложения.

1. INFO

 — информационные сообщения.

1. WARNING

 — предупреждения.

1. ERROR

 — сообщения об ошибках.

1. CRITICAL

 — критические сообщения.

Уровень, назначенный логеру, действует как порог маркера серьезности. Любые попытки логирования сообщения с более низкой серьезностью игнорируются. Если уровень логирования установлен на

WARNING

, логер будет выводить сообщения с серьезностью

WARNING

 и выше. Сообщения с более низкой серьезностью, такие как

DEBUG

 и

INFO

, игнорируются.

Для логирования сообщений различного уровня в библиотеке

logging

 предусмотрены соответствующие методы:

1. Для сообщений уровня

DEBUG

 применяется метод

debug()

.

1. Для сообщений уровня

INFO

 — метод

info()

.

1. Для сообщений уровня

WARNING

 — метод

warning()

.

1. Для сообщений уровня

ERROR

 — метод

error()

.

1. Для сообщений уровня

CRITICAL

 — метод

critical()

.

app\_logger.debug("Это сообщение уровня DEBUG")

app\_logger.info("Это сообщение уровня INFO")

app\_logger.warning("Это сообщение уровня WARNING")

app\_logger.error("Это сообщение уровня ERROR")

app\_logger.critical("Это сообщение уровня CRITICAL")

Эти методы добавляют сообщения в лог на соответствующем уровне:

2023-10-17 14:52:29,394 DEBUG: Это сообщение уровня DEBUG

2023-10-17 14:52:29,394 INFO: Это сообщение уровня INFO

2023-10-17 14:52:29,394 WARNING: Это сообщение уровня WARNING

2023-10-17 14:52:29,394 ERROR: Это сообщение уровня ERROR

2023-10-17 14:52:29,394 CRITICAL: Это сообщение уровня CRITICAL

Рассмотрим на примере, как происходит логирование работы программы на разных уровнях:

# Импортируем модуль logging

import logging

# Получаем корневой логер

logger = logging.getLogger()

# Логируем сообщение уровня ERROR

logger.error("Это ошибка")

# Получаем логер с определенным именем

named\_logger = logging.getLogger("mylogger")

# Логируем сообщение уровня CRITICAL

named\_logger.critical("Очень критично")

Разберем по шагам, что происходит в примере выше:

1. **Импорт библиотеки**

logging

.

Сначала мы импортируем библиотеку

logging

. Это необходимо, чтобы использовать функциональность логирования, предоставляемую этой библиотекой.

1. **Получение корневого логера:**

logger = logging.getLogger()

Здесь вызывается метод

getLogger()

 без аргументов, что возвращает корневой логер (root logger).

**Корневой логер** является основным логером, который существует по умолчанию и используется, если явно не указан другой логер. Корневой логер задается общими настройками для всего приложения.

1. **Логирование сообщения уровня ERROR:**

logger.error("Это ошибка")

В этом шаге корневой логер записывает сообщение уровня ERROR. Сообщения этого уровня обычно обозначают ошибки, которые нужно немедленно исправить. По умолчанию сообщения уровня ERROR и выше будут отображены в консоли.

1. **Получение именованного логера:**

named\_logger = logging.getLogger("mylogger")

В этом шаге мы создаем логер с именем

"mylogger"

.

**Именованные логеры** позволяют разделять и логировать сообщения по различным частям или модулям вашего приложения. Имя логера помогает в организованном и структурированном подходе к логированию.

1. **Логирование сообщения уровня CRITICAL:**

named\_logger.critical("Очень критично")

Этот код записывает сообщение уровня CRITICAL с использованием именованного логера

"mylogger"

.

В примере мы работали с методом

getLogger()

. Библиотека

logging

 предоставляет два варианта метода

getLogger()

. Подробнее об этом мы поговорим далее.

# 4. Компоненты логирования

Модуль логирования в Python состоит из четырех компонентов:

1. Logger

 — это объект, который пишет логи. Он принимает сообщения от программы и создает для них записи лога. Он также знает, какого уровня должны быть логи.

**Уровень лога** — это важность сообщения.

Например, если сообщение говорит об ошибке, то уровень лога —

ERROR

. Если сообщение говорит о том, что программа работает нормально, то уровень лога —

INFO

.

Логер нужен для того, чтобы писать логи.

1. Handler

 — это объект, который отправляет логи. Он принимает записи лога от логера и решает, куда их отправить. Например, он может отправить логи в файл на компьютере или на экран консоли.

Хендлер нужен для того, чтобы выбирать место для логов.

1. Filter

 — это объект, который фильтрует логи. Он принимает записи лога от логера или хендлера и решает, какие из них нужно обработать, а какие нет. Например, он может пропускать логи ниже определенного уровня или по определенному слову.

Фильтр нужен для того, чтобы отсеивать лишние логи.

1. Formatter

 — это объект, который форматирует логи. Он принимает записи лога от хендлера и решает, как они должны выглядеть. Например, он может добавить дату, время, уровень и другие данные к логам.

Форматер нужен для того, чтобы делать логи красивыми и понятными.

Чтобы разобраться в работе логера, хендлера, фильтра и форматера, вам потребуется знать, что такое

LogRecord

.

LogRecord

 — это объект, созданный

Logger

, который содержит всю информацию о сообщении лога. Он содержит такие атрибуты, как имя логера, уровень логирования, время, сообщение и многое другое.

LogRecord

 передается в

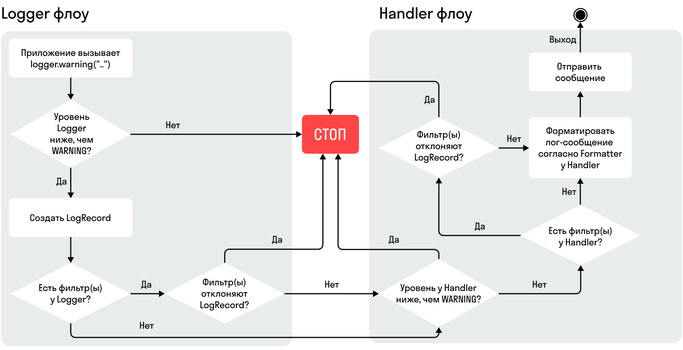
Handler

, который затем отправляет его в соответствующем направлении для записи.

Теперь, когда вы знаете, что такое логер, хендлер, фильтр и форматер, вы можете посмотреть, как они работают вместе. На схеме ниже показано, что происходит, когда программа хочет записать лог на уровне

WARNING

. Это значит, что в программе возникло что-то необычное или потенциально опасное для ее работы, но не критичное, что требует немедленного реагирования.



### Лучшая практика

Предпочтительнее установить уровень логирования на логере и отказаться от установки уровня логирования на обработчике. Возможность установки уровня логирования на обработчике наиболее полезна в том случае, если у логера есть несколько присоединенных обработчиков и мы хотим, чтобы каждый обработчик регистрировал только подмножество сообщений, проходящих через логер.

Например, если уровень логера установлен на

INFO

 и к нему прикреплены два обработчика — обработчик потока, регистрирующий в консоль, и обработчик потока, регистрирующий в файл, — а мы хотим регистрировать сообщения об ошибках только в файле, то оставляем уровень на обработчике, регистрирующем в консоль как есть, но устанавливаем уровень на обработчике, регистрирующем в файл, на

ERROR

.

# 5. Логеры

Логер играет центральную роль в системе логирования и позволяет организовать сбор, фильтрацию и запись логов.

Logger

 (регистратор) — это объект, предназначенный для записи логов. Он получает сообщения от приложения, создает для каждого сообщения объект

LogRecord

, назначает ему уровень важности (

DEBUG

,

INFO

,

WARNING

 и т. д.) и передает его в обработчики —

Handlers

.

## Создание и получение логеров

В библиотеке

logging

 можно создать и получить логер с помощью функции

getLogger()

. Поведение функции различается в зависимости от того, передан ли ей аргумент:

import logging

# Получение корневого логера

root\_logger = logging.getLogger()

# Создание и получение именованного логера

app\_logger = logging.getLogger("my\_application")

Если

getLogger()

 вызывается без аргументов, то возвращается корневой логер (root logger). Если же функция вызывается с именем логера, которое еще не было ранее использовано, то создается новый логер. Если логер с таким именем уже существует, возвращается существующий логер.

В качестве аргумента для функции

getLogger()

 можно использовать параметр

\_\_name\_\_

, который установит имя логера равное имени модуля.

Пример:

import logging

# Получение корневого логера

root\_logger = logging.getLogger()

# Создание и получение именованного логера

app\_logger = logging.getLogger(\_\_name\_\_)

\_\_name\_\_

 — это специальная переменная, которая содержит имя модуля, из которого она вызвана. Если модуль является главным, то есть запущен напрямую, а не импортирован, то значение переменной будет

\_\_main\_\_

, в противном случае — имя модуля (имя файла без

.py

).

## Установка уровня логирования

Логер позволяет установить уровень важности сообщений, которые будут записываться. Это делается с помощью метода

setLevel()

:

import logging

app\_logger = logging.getLogger("my\_application")

app\_logger.setLevel(logging.DEBUG)

# 6. Хендлеры

Handler

**(обработчик)** — это объект, который определяет, куда и как выводить логи.

* Библиотека

logging

 предоставляет несколько типов хендлеров, например для записи в файл и вывода на консоль (

StreamHandler

,

FileHandler

) и другие.

* Логер может иметь один или несколько обработчиков (

Handlers

), которые определяют, куда отправлять сообщения логов. Вы можете добавлять и настраивать обработчики в зависимости от ваших потребностей с помощью функции

addHandler()

.

Рассмотрим различные типы хендлеров.

## StreamHandler

StreamHandler

 — тип хендлера, который используется для вывода логов в консоль.

Например:

import logging

logger = logging.getLogger(\_\_name\_\_)

# Создаем хендлер для вывода в консоль

console\_handler = logging.StreamHandler()

logger.addHandler(console\_handler)

logger.setLevel(logging.DEBUG)

logger.debug('Debug message')

Разберем, что происходит в этом коде:

1. Подключаем модуль

logging

:

import logging

1. Создаем логер с именем текущего модуля

\_\_name\_\_

:

logger = logging.getLogger(\_\_name\_\_)

1. Создаем обработчик, который будет выводить лог-сообщения в консоль:

console\_handler = logging.StreamHandler()

1. Добавляем обработчик к логеру. Это означает, что все сообщения, отправленные логеру, будут обрабатываться этим обработчиком и выводиться в консоль:

logger.addHandler(console\_handler)

1. Устанавливаем уровень логирования для логера на

DEBUG

. Логер будет обрабатывать все сообщения уровня

DEBUG

 и выше (

INFO

,

WARNING

,

ERROR

,

CRITICAL

):

logger.setLevel(logging.DEBUG)

1. Отправляем сообщение уровня

DEBUG

 —

'Debug message'

. Поскольку уровень логера установлен на

DEBUG

, это сообщение будет обработано и выведено в консоль.

logger.debug('Debug message')

## FileHandler

FileHandler

 — тип хендлера, который используется для вывода логов в файл.

Как правило, каждый логер имеет свой собственный файл лога. Например, логер

app\_logger

 может выводить логи в файл

app.log

, а логер

db\_logger

 — в файл

db.log

.

Рассмотрим пример:

import logging

logger = logging.getLogger(\_\_name\_\_)

# Создаем хендлер для вывода в файл

file\_handler = logging.FileHandler('example.log')

logger.addHandler(file\_handler)

logger.setLevel(logging.DEBUG)

logger.debug('Debug message')

Разберем, что происходит в этом коде:

1. Подключаем модуль

logging

:

import logging

1. Создаем логер с именем текущего модуля

\_\_name\_\_

:

logger = logging.getLogger(\_\_name\_\_)

1. Создаем хендлер для вывода лог-сообщений в файл

example.log

:

file\_handler = logging.FileHandler('example.log')

1. Добавляем хендлер к логеру. Это означает, что все сообщения, отправленные логеру, будут обрабатываться этим хендлером и записываться в файл.

logger.addHandler(file\_handler)

1. Устанавливаем уровень логирования для логера на

DEBUG

. Логер будет обрабатывать все сообщения уровня

DEBUG

 и выше (

INFO

,

WARNING

,

ERROR

,

CRITICAL

):

logger.setLevel(logging.DEBUG)

1. Отправляем сообщение уровня

DEBUG

 —

'Debug message'

. Поскольку уровень логирования установлен на

DEBUG

, это сообщение будет обработано и записано в файл

example.log

.

logger.debug('Debug message')

## Другие хендлеры

Кроме указанных выше, библиотека

logging

 предоставляет и другие типы хендлеров:

SMTPHandler

,

SocketHandler

,

SysLogHandler

 и др. Каждый тип хендлера имеет свои особенности и может быть настроен по-разному в зависимости от требований приложения.

О работе других хендлеров вы можете прочитать в документации (на английском языке): <https://docs.python.org/3/howto/logging.html#useful-handlers>.

# 7. Форматеры

Библиотека

logging

 позволяет настраивать формат вывода логов.

Formatter

 — это тот компонент библиотеки

logging

, который определяет, как выводить сообщения логов. Он связан только с обработчиком и не влияет на логирование сообщений.

Форматер использует специальные строки формата для определения того, как выводить сообщения логов. Строка формата может содержать текстовые сообщения и специальные метки для вставки информации из

LogRecord

. Специальные метки начинаются с символа

%

 и могут содержать различные опции форматирования. Например,

%(asctime)s

 означает, что мы хотим вставить дату и время события логирования в виде строки (указание символа

s

).

Рассмотрим пример:

import logging

logger = logging.getLogger(\_\_name\_\_)

file\_handler = logging.FileHandler('example.log')

file\_formatter = logging.Formatter('%(asctime)s %(levelname)s: %(message)s')

file\_handler.setFormatter(file\_formatter)

logger.addHandler(file\_handler)

logger.setLevel(logging.DEBUG)

logger.debug('Debug message')

logger.info('Info message')

logger.warning('Warning message')

logger.error('Error message')

logger.critical('Critical message')

Как видно из примера, мы создаем логер с именем модуля

\_\_name\_\_

, затем создаем хендлер

FileHandler

 для вывода логов в файл

example.log

. Создаем форматер

Formatter

 для форматирования вывода используемого хендлера. Затем устанавливаем созданный форматер для хендлера и добавляем хендлер в логер. Устанавливаем уровень логирования на

DEBUG

.

После этого мы выводим сообщения разных уровней — от

DEBUG

 до

CRITICAL

.

Когда мы запустим код, то увидим, что файл

example.log

 был создан в рабочей директории и содержит подобный текст:

2023-10-17 14:52:29,394 DEBUG: Debug message

2023-10-17 14:52:29,394 INFO: Info message

2023-10-17 14:52:29,394 WARNING: Warning message

2023 -10-17 14:52:29,394 ERROR: Error message

2023-10-17 14:52:29,394 CRITICAL: Critical message

Форматирование позволяет задать, как будут выглядеть логи, когда мы их записываем или выводим.

В примере мы использовали формат

'%(asctime)s %(levelname)s: %(message)s'

. Это означает, что мы хотим, чтобы логи выглядели так:

[дата и время] [уровень логирования]: [текст сообщения]

Например, если мы записываем лог на уровне

WARNING

 с сообщением

File not found

, то он будет выглядеть так:

2023-10-01 12:34:56 WARNING: File not found

Кроме того, можно добавлять в форматирование другие параметры, такие как имя логера, номер строки, имя функции и т. д. Вот полный список параметров для форматирования:

* %(asctime)s

 — дата и время события логирования.

* %(name)s

 — имя логера.

* %(levelname)s

 — уровень логирования.

* %(message)s

 — текст сообщения.

* %(filename)s

 — имя файла, в котором произошло событие.

* %(funcName)s

 — имя функции, в которой произошло событие.

* %(lineno)d

 — номер строки, в которой произошло событие.

* %(process)d

 — ID процесса.

* %(thread)d

 — ID потока.

Например, если мы хотим добавить имя логера и номер строки в формат сообщения, то мы можем изменить формат следующим образом:

import logging

logger = logging.getLogger(\_\_name\_\_)

console\_handler = logging.StreamHandler()

console\_formatter = logging.Formatter('%(asctime)s %(levelname)s %(name)s %(lineno)d: %(message)s')

console\_handler.setFormatter(console\_formatter)

logger.addHandler(console\_handler)

logger.setLevel(logging.DEBUG)

logger.debug('Debug message')

Когда мы запустим код, то увидим следующий вывод:

2023-10-17 14:52:29,394 DEBUG \_\_main\_\_ 4: Debug message

## Пример логирования в реальной программе

Для демонстрации возьмем пример программы, которая выполняет несколько основных операций: авторизацию пользователя, выполнение операций с базой данных и обработку ошибок.

**Шаг 1. Импорт библиотеки logging и создание логера:**

import logging

# Основная конфигурация logging

logging.basicConfig(level=logging.DEBUG,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

filename='application.log', # Запись логов в файл

filemode='w') # Перезапись файла при каждом запуске

# Создаем логеры для различных компонентов программы

auth\_logger = logging.getLogger('app.auth')

db\_logger = logging.getLogger('app.database')

main\_logger = logging.getLogger('app.main')

Подробнее о logging.basicConfig()

**Шаг 2. Логирование отдельных модулей.**

Функция авторизации пользователя:

def login(username, password):

# Записываем информацию о попытке входа в лог

auth\_logger.info(f'Попытка входа для пользователя: {username}')

# Проверяем правильность введенного имени пользователя и пароля

if username == "admin" and password == "secret":

# Если проверка успешна, записываем сообщение об успешной авторизации

auth\_logger.info('Успешная авторизация')

return True

else:

# Если проверка неуспешна, записываем предупреждение о неудачной попытке входа

auth\_logger.warning('Неудачная попытка входа')

return False

* Первый вызов метода

auth\_logger.info()

 — записывает информационное сообщение о попытке входа. Это сообщение будет полезно для отслеживания действий пользователей.

* Второй вызов метода

auth\_logger.info()

 — записывает информационное сообщение об успешной авторизации, если имя пользователя и пароль совпадают с ожидаемыми значениями.

* auth\_logger.warning()

 — записывает предупреждение, если имя пользователя и/или пароль неверны.

Функции работы с базой данных:

data = {} # Используем словарь как простую базу данных

def insert(key, value):

# Записываем информацию о вставке данных

db\_logger.info(f'Вставка данных: {key} = {value}')

# Вставляем данные в словарь

data[key] = value

def select(key):

# Пытаемся получить значение по ключу из словаря

value = data.get(key)

# Проверяем, найдено ли значение

if value:

# Если значение найдено, записываем информацию об успешном получении данных

db\_logger.info(f'Получены данные: {key} = {value}')

else:

# Если значение не найдено, записываем предупреждение о ненайденных данных

db\_logger.warning(f'Данные с ключом {key} не найдены')

return value

* db\_logger.info()

 — записывает информационное сообщение о вставке данных в базу данных. Это помогает отслеживать изменения данных.

* data[key] = value

 — фактически вставляет данные в словарь, который мы используем в качестве простой базы данных.

* db\_logger.info()

 — записывает информационное сообщение при успешном получении значения из базы данных.

* db\_logger.warning()

 — записывает предупреждение, если значение по указанному ключу не найдено. Это помогает отслеживать ошибки и отсутствующие данные в базе данных.

Основная функция:

def main():

try:

# Записываем сообщение о запуске приложения

main\_logger.info('Запуск приложения')

# Попытка авторизации

user\_logged\_in = login("admin", "secret")

if user\_logged\_in:

# Если авторизация успешна, работаем с базой данных

insert("user\_id", "12345")

select("user\_id")

select("non\_existing\_id")

else:

# Если авторизация не успешна, записываем предупреждение и прекращаем работу

main\_logger.warning("Неудачная авторизация. Прекращение работы")

except Exception as e:

# Записываем ошибку, если произошло исключение во время выполнения программы

main\_logger.error(f'Произошла ошибка: {e}', exc\_info=True)

finally:

# Записываем сообщение о завершении работы приложения

main\_logger.info('Завершение работы приложения')

if \_\_name\_\_ == "\_\_main\_\_":

main()

* main\_logger.info()

 — записывает информационное сообщение о запуске приложения.

* main\_logger.warning()

 — записывает предупреждение, если авторизация не удалась, и приложение завершает работу.

* main\_logger.error()

 — записывает сообщение об ошибке, если во время выполнения программы произошло исключение. Параметр

exc\_info=True

 добавляет информацию об исключении в лог.

* main\_logger.info()

 — записывает информационное сообщение о завершении работы приложения.

### Что будет в логах

В результате выполнения этой программы файл

application.log

 может выглядеть следующим образом:

2023-10-17 15:35:23,456 - app.main - INFO - Запуск приложения

2023-10-17 15:35:23,457 - app.auth - INFO - Попытка входа для пользователя: admin

2023-10-17 15:35:23,457 - app.auth - INFO - Успешная авторизация

2023-10-17 15:35:23,458 - app.database - INFO - Вставка данных: user\_id = 12345

2023-10-17 15:35:23,459 - app.database - INFO - Получены данные: user\_id = 12345

2023-10-17 15:35:23,459 - app.database - WARNING - Данные с ключом non\_existing\_id не найдены

2023-10-17 15:35:23,460 - app.main - INFO - Завершение работы приложения

Этот пример показывает, как можно использовать библиотеку

logging

 для создания структурированных и полезных логов с минимальными усилиями. Логи помогут быстро понять, что происходит в приложении, и упростят процесс отладки и мониторинга.

# 8. Логирование на практике

Требуется разработать приложение, которое будет собирать новости с сайта [newsapi.org](http://newsapi.org/) с помощью API.

1. При запуске приложение получает текущую дату и извлекает статьи, опубликованные на сайте за этот день. Все собранные данные сохраняются в JSON-файл.
2. В файле содержится только основная информация:
   * заголовок статьи,
   * имя автора,
   * краткое описание,
   * ссылка на полную статью.
3. В приложении предусмотрены возможности указать ключевые слова, которые должны присутствовать в статье, а также список слов, наличие которых исключает статью из результатов.
4. Название JSON-файла генерируется автоматически:
   * сначала указывается дата запроса,
   * затем добавляются ключевые слова, указанные в запросе, разделенные нижним подчеркиванием.
5. Настройте логирование работы программы в файл.